# Advanced Rcpp I

Cecina Babich Morrow

2024-05-13

## Local polynomial regression

In this portfolio, we will demonstrate how to perform local polynomial regression both in `R` directly and by using `RcppArmadillo`.

### Load the data

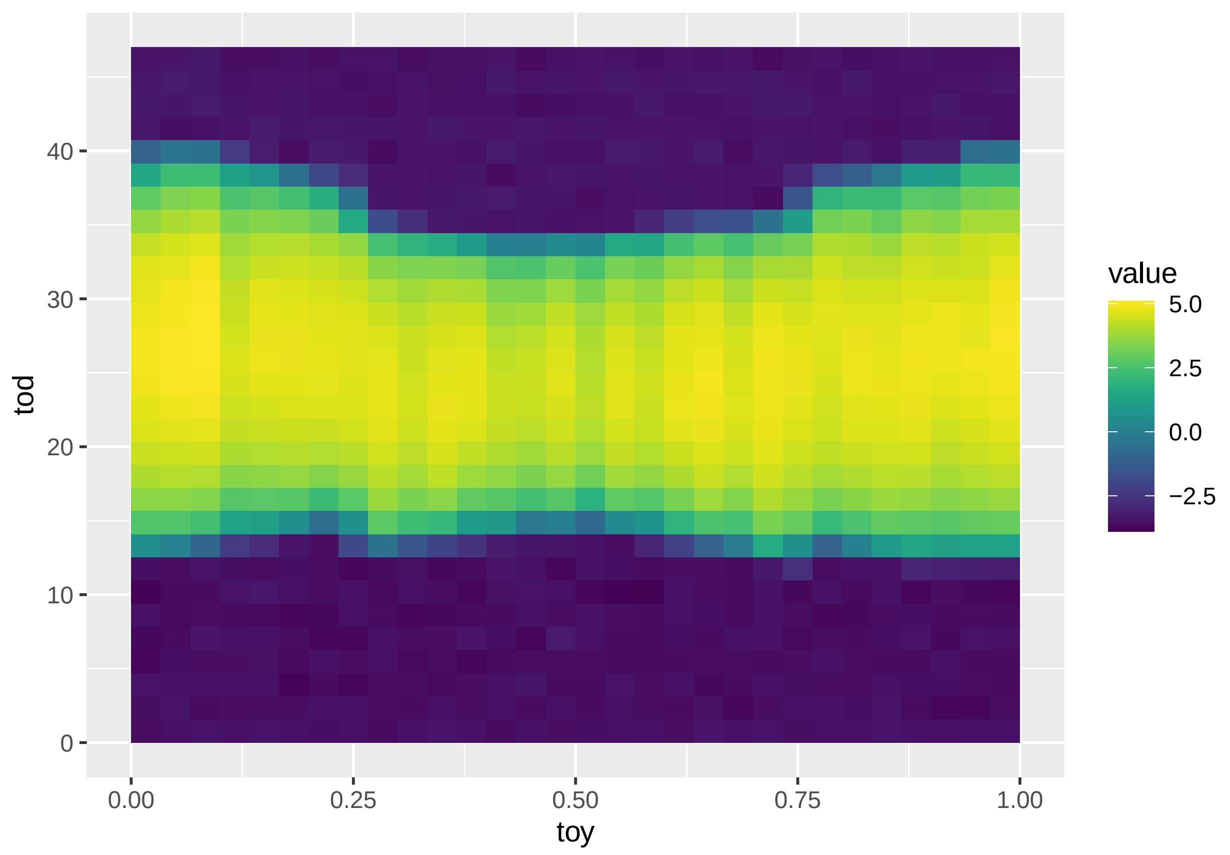
We have the following data set on solar electricity production in Sydney, Australia:

```r
library(here)
library(tidyverse)
load(here("portfolios/03_advanced_rcpp_1/data/solarAU.RData"))
head(solarAU)
```

```
##       prod          toy tod
## 8832 0.019 0.000000e+00   0
## 8833 0.032 5.708088e-05   1
## 8834 0.020 1.141618e-04   2
## 8835 0.038 1.712427e-04   3
## 8836 0.036 2.283235e-04   4
## 8837 0.012 2.854044e-04   5
```

We will add a column for the log-transformed production:

```r
solarAU$logprod <- log(solarAU$prod+0.01)
# Visualize log-production
library(viridis)
ggplot(solarAU,
       aes(x = toy, y = tod, z = logprod)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))
```



## Q1: Linear regression model

We want to model $\mathbb{E}(y|\mathbf{x})$ where $y$ is the log-production and $\mathbf{x} = \{\text{tod}, \text{toy}\}$. We will use a polynomial regression model of degree 2:

$$\mathbb{E}(y|\mathbf{x}) = \beta_0 + \beta_1 \text{tod} + \beta_2 \text{tod}^2 + \beta_3 \text{toy} + \beta_4 \text{toy}^2 = \tilde{\mathbf{x}}^\top \beta$$

where $\tilde{\mathbf{x}} = \{1, \text{tod}, \text{tod}^2, \text{toy}, \text{toy}^2\}$.

Using R, we would fit the model as follows:

```r
fit <- lm(logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
summary(fit)
```

```
##
## Call:
## lm(formula = logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.1623 -1.5447  0.6765  1.6280  4.8534
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6.263e+00  6.422e-02  -97.52   <2e-16 ***
## tod          8.644e-01  4.513e-03  191.53   <2e-16 ***
## I(tod^2)    -1.758e-02  9.286e-05 -189.28   <2e-16 ***
## toy         -5.918e+00  2.207e-01  -26.81   <2e-16 ***
## I(toy^2)     6.143e+00  2.138e-01   28.74   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.106 on 17467 degrees of freedom
## Multiple R-squared:  0.6838, Adjusted R-squared:  0.6838
## F-statistic:  9445 on 4 and 17467 DF,  p-value: < 2.2e-16
```

We now want to use `RcppArmadillo` to fit a linear regression model by solving

$$\hat{\beta} = \arg\min_{\beta} \|y - \mathbf{X}\beta\|^2$$

We can define $\mathbf{X}$ and $y$ in R by:

```
X <- with(solarAU, cbind(1, tod, tod^2, toy, toy^2))
y <- solarAU$logprod
```

We can write a function to fit the model using QR decomposition with `RcppArmadillo`:

```
library(Rcpp)
sourceCpp(code = '
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace arma;

// [[Rcpp::export(name = "armadillo_lm")]]
vec armadillo_lm(mat& X, vec& y) {
  mat Q;
  mat R;

  qr_econ(Q, R, X);
  vec beta = solve(R, (trans(Q) * y));
  return beta;
}
')
```

We will compare the results and performance with the `lm` function in R:

```
arma_lm_beta <- armadillo_lm(X,y)
max(abs(coef(fit) - arma_lm_beta)) # results are the same
```

```
## [1] 4.360956e-13
```

```
library(microbenchmark)
lm_R <- function() lm(logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
lm_arma <- function() armadillo_lm(X,y)
microbenchmark(lm_R(), lm_arma(), times = 500)
```

```
## Unit: microseconds
##       expr      min       lq     mean   median       uq       max neval
##     lm_R() 3514.268 3908.512 5152.841 4154.327 4708.030 86359.410   500
##  lm_arma()  984.613 1081.579 1145.280 1131.156 1187.418  2749.831   500
```

The implementation in `RcppArmadillo` is over 4 times faster on average than the `lm` function in R.

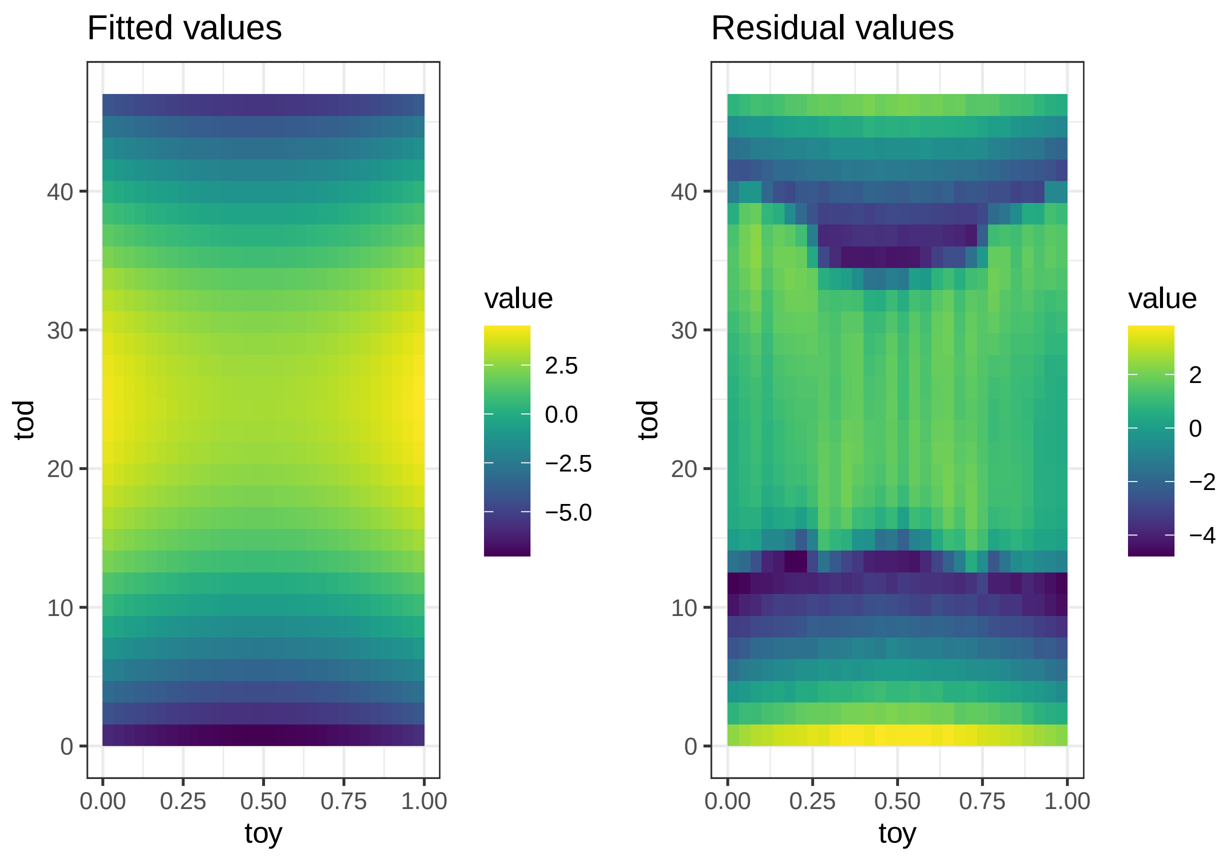
We can now see if the model is a good fit for the data:

```
library(gridExtra)
```

```r
solarAU <- solarAU %>%
  mutate(lm_fitted = fit$fitted.values,
         lm_resids = fit$residuals)

fitted_plot <- ggplot(solarAU,
                      aes(x = toy, y = tod, z = lm_fitted)) +
  stat_summary_2d() +
  scale_fill_gradientn(colours = viridis(50)) +
  theme_bw() +
  labs(title = "Fitted values")

resid_plot <- ggplot(solarAU,
                     aes(x = toy, y = tod, z = lm_resids)) +
  stat_summary_2d() +
  scale_fill_gradientn(colours = viridis(50)) +
  theme_bw() +
  labs(title = "Residual values")
grid.arrange(fitted_plot, resid_plot, ncol = 2)
```



The plot of residuals shows a non-linear pattern, suggesting that the model is not a good fit for the data.

## Q2: Local least squares regression

We want to try to address this non-linear pattern in the residuals. One possible solution is to use local regression, which fits separate regression models to different subsets of the data. We will now let the regression coefficients be a function of the covariates $\mathbf{x}$, i.e. $\hat{\beta} = \hat{\beta}(\mathbf{x})$. For a given $\mathbf{x}_0$,

$$\hat{\beta}(\mathbf{x}_0) = \arg\min_{\beta} \sum_{i=1}^{n} \kappa_{\mathbf{H}}(\mathbf{x}_0 - \mathbf{x}_i)(y_i - \tilde{\mathbf{x}}_i^\top \beta)^2$$

where $\kappa_H$ is a density kernel with positive definite bandwidth matrix $\mathbf{H}$.

We can implement a function in R to do this using the Gaussian kernel:

```r
library(mvtnorm)
lmLocal <- function(y, x0, X0, x, X, H){
  w <- dmvnorm(x, x0, H)
  fit <- lm(y ~ -1 + X, weights = w)
  return( t(X0) %*% coef(fit) )
}
```

The `lmLocal` function takes the following inputs:

- `y`: response variable
- `x0`: location of the prediction
- `X0` : design matrix at the prediction location
- `x` : locations of the data points
- `X` : design matrix at the data points
- `H` : bandwidth matrix

It returns the fitted values at the prediction location.

We can test this model using a subset of 2000 data points (since fitting it on all 17472 data points would be computationally costly):

```r
set.seed(123)
n <- nrow(X)
nsub <- 2e3
# Get 2000 random indices
sub <- sample(1:n, nsub, replace = FALSE)

y <- solarAU$logprod
solarAU_sub <- solarAU[sub, ]
x <- as.matrix(solarAU[c("tod", "toy")])
x0 <- x[sub, ]
X0 <- X[sub, ]
# Using the following H
H <- diag(c(1, 0.1)^2)

# Obtain estimates at each subsampled location
# And check how long that takes
tictoc::tic()
predLocal <- sapply(1:nsub, function(ii){
  lmLocal(y = y, x0 = x0[ii, ], X0 = X0[ii, ], x = x, X = X, H = H)
})
tictoc::toc()
```

```
## 24.036 sec elapsed
```

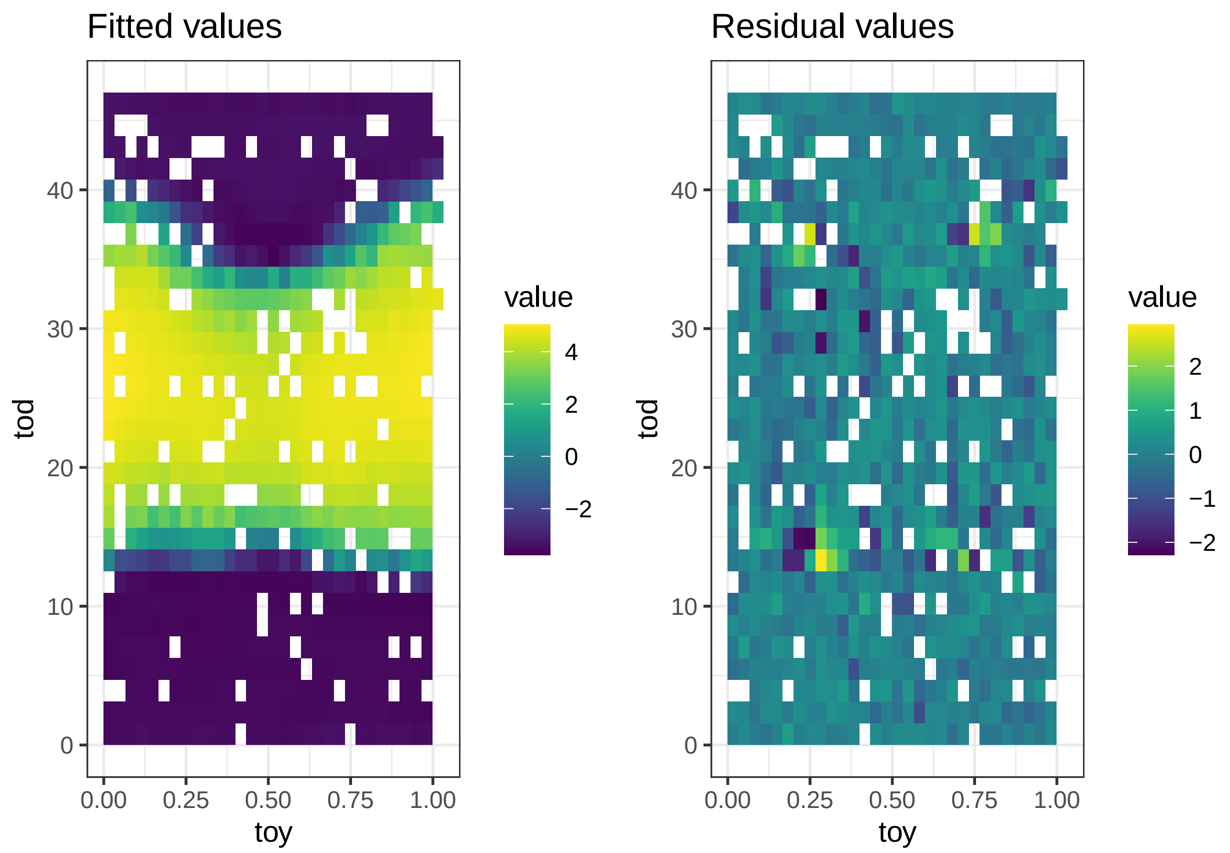
We can see this is a fairly slow process. Let's see how the fit / residuals look:

```r
solarAU_sub <- solarAU_sub %>%
  mutate(lmLocal_fitted = predLocal,
         lmLocal_resids = logprod - predLocal)
```

```r
fitted_plot <- ggplot(solarAU_sub,
                      aes(x = toy, y = tod, z = lmLocal_fitted)) +
  stat_summary_2d() +
  scale_fill_gradientn(colours = viridis(50)) +
  theme_bw() +
  labs(title = "Fitted values")

resid_plot <- ggplot(solarAU_sub,
                     aes(x = toy, y = tod, z = lmLocal_resids)) +
  stat_summary_2d() +
  scale_fill_gradientn(colours = viridis(50)) +
  theme_bw() +
  labs(title = "Residual values")

grid.arrange(fitted_plot, resid_plot, ncol = 2)
```



We no longer have a pattern in the residuals, and our fitted values closely resemble our original data.

Let's see if we can implement the local least squares in RcppArmadillo to get these good results without the high computational cost. See here for the `armadillo_lm_local.cpp` code file, which depends on two functions included in a header file `armadillo_lm_funcs.h`.

```r
sourceCpp(file = 'armadillo_lm_local.cpp')

arma_predLocal <- armadillo_lm_local(y, x0, X0, x, X, H)
max(abs(predLocal - arma_predLocal)) # results are the same
```

```
## [1] 4.085621e-13
```

```
predLocal_R <- function() sapply(1:nsub, function(ii){
  lmLocal(y = y, x0 = x0[ii, ], X0 = X0[ii, ], x = x, X = X, H = H)
})
predLocal_arma <- function() armadillo_lm_local(y, x0, X0, x, X, H)
microbenchmark(predLocal_R(), predLocal_arma(), times = 5) # only 5 runs because R is super slow
```

```
## Unit: seconds
##              expr      min       lq     mean   median       uq      max neval
##     predLocal_R() 23.42875 25.39191 26.50795 25.41072 26.68802 31.62032     5
##  predLocal_arma() 15.25194 15.25607 18.02687 15.31134 15.53078 28.78423     5
```

The `RcppArmadillo` implementation is once again significantly faster than the R implementation.

## Q3: Choosing bandwidth with cross-validation

We will now use cross-validation to choose the bandwidth matrix **H**. See here for the `armadillo_cv_H.cpp` code file. We will perform 5-fold cross-validation.

```
sourceCpp(file = 'armadillo_cv_H.cpp')

# Set of H options for cross-validation
H_options <- list(diag(c(1, 0.1)^2),
                  diag(c(1, 0.01)^2),
                  diag(c(0.9, 0.1)^2))

tictoc::tic()
arma_predLocal_cv <- armadillo_cv_H(y, x0, X0, x, X, H_options, 5)
tictoc::toc()
```

```
## 23.29 sec elapsed
```

We can check which bandwidth matrix **H** was chosen:

```
arma_predLocal_cv$H
```

```
##      [,1]  [,2]
## [1,]    1 0e+00
## [2,]    0 1e-04
```

We can now see the fits and residuals from using cross-validation to choose the bandwidth matrix:

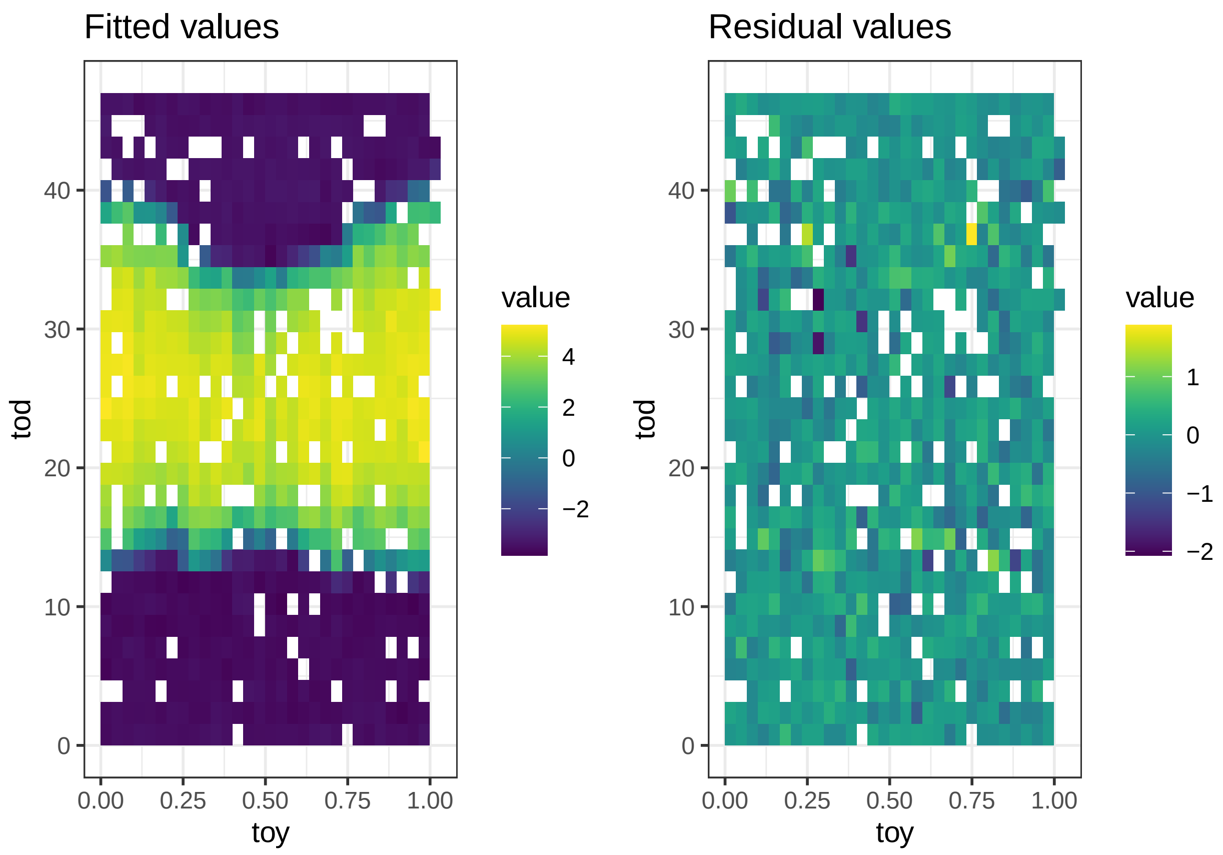
```
solarAU_sub <- solarAU_sub %>%
  mutate(lmLocal_cv_fitted = arma_predLocal_cv$fitted[,1],
         lmLocal_cv_resids = logprod -  arma_predLocal_cv$fitted[,1])

fitted_plot_cv <- ggplot(solarAU_sub,
                      aes(x = toy, y = tod, z = lmLocal_cv_fitted)) +
  stat_summary_2d() +
  scale_fill_gradientn(colours = viridis(50)) +
  theme_bw() +
  labs(title = "Fitted values")

resid_plot_cv <- ggplot(solarAU_sub,
                     aes(x = toy, y = tod, z = lmLocal_cv_resids)) +
  stat_summary_2d() +
  scale_fill_gradientn(colours = viridis(50)) +
```

```r
  theme_bw() +
  labs(title = "Residual values")

grid.arrange(fitted_plot_cv, resid_plot_cv, ncol = 2)
```



It's hard to visually see the difference between the fits and residuals from the cross-validated bandwidth matrix and the original bandwidth matrix. We can compare the residuals from the two methods:

```r
# MSE from original bandwidth matrix
mse_orig <- mean(solarAU_sub$lmLocal_resids^2)
mse_orig
```

```
## [1] 0.3801064
```

```r
# MSE from cross-validated bandwidth matrix
mse_cv <- mean(solarAU_sub$lmLocal_cv_resids^2)
mse_cv
```

```
## [1] 0.2243705
```

We can see that the mean squared error is lower when using the cross-validated bandwidth matrix.

## Package creation

I created an R package called `armadilLocalregression` using the functions created above (the package is available on GitHub here). Creating a package using `RcppArmadillo` had some additional intricacies compared to creating an `R` package without any `C++` code. After creating the package and adding the `R` functions `lmLocal` and `predLocal` (as in exercise Q2), I needed to run `RcppArmadillo.package.skeleton()` in order to adapt my package structure.

I then added functions using `RcppArmadillo` using the `use_rcpp_armadillo()` function from the `usethis` package. The `roxygen` comments for these functions need to start with `//'`, rather than `#'` as in the `R` functions. The functions `armadillo_lm` and `armadillo_lm_local` are both called by other functions, so I put them into a header file `armadillo_lm_funcs.h`. I was still able to add `roxygen` comments to this file.

We can install the package from GitHub and test the functions:

```r
devtools::install_github("babichmorrowc/armadilLocalregression")
```

```
## Skipping install of 'armadilLocalregression' from a github remote, the SHA1 (399d9787) has not chang
##   Use `force = TRUE` to force installation
```

```r
library(armadilLocalregression)
```

```
##
## Attaching package: 'armadilLocalregression'

## The following objects are masked _by_ '.GlobalEnv':
##
##     armadillo_cv_H, armadillo_lm, armadillo_lm_local, lmLocal
```

```r
# Testing armadillo_lm
pkg_arma_lm_beta <- armadilLocalregression::armadillo_lm(X,y)
# Check that the package version of the function gives the same answer
all(arma_lm_beta == pkg_arma_lm_beta)
```

```
## [1] TRUE
```

```r
# Testing predLocal
pkg_predLocal <- armadilLocalregression::predLocal(y, x0, X0, x, X, H)
# Check that the package version of the function gives the same answer
all(predLocal == pkg_predLocal)
```

```
## [1] TRUE
```

```r
# Testing armadillo_lm_local
pkg_predLocal_arma <- armadilLocalregression::armadillo_lm_local(y, x0, X0, x, X, H)
# Check that the package version of the function gives the same answer
all(arma_predLocal == pkg_predLocal_arma)
```

```
## [1] TRUE
```

```r
# Testing armadillo_cv_H
pkg_predLocal_cv <- armadilLocalregression::armadillo_cv_H(y, x0, X0, x, X, H_options, 5)
# Check that the package version of the function gives the same answer
all(arma_predLocal_cv$H == pkg_predLocal_cv$H)
```

```
## [1] TRUE
```

```r
all(arma_predLocal_cv$fitted == pkg_predLocal_cv$fitted)
```

```
## [1] TRUE
```

The package functions operate in the same way as the functions written in the portfolio.

I set up GitHub Actions to check the package on every push to the repository. Although the package does have simple examples for the functions, I would like to add unit testing in the future via the `testthat` package to ensure that the functions work as expected, particularly on datasets other than the solar electricity production data for which they were developed.